Tuning Pipelined Scientific Data Analyses for Efficient Multicore Execution

Andre Pereira
Department of Informatics
LIP and University of Minho
Braga, Portugal
Email: ampereira90@gmail.com

Antonio Onofre
Department of Physics
LIP and University of Minho
Braga, Portugal
Email: Antonio.Onofre@cern.ch

Alberto Proenca
Department of Informatics
University of Minho
Braga, Portugal
Email: aproenca@di.uminho.pt

***Abstract*—Scientific data analyses often apply a pipelined sequence of computational tasks to independent datasets. Each task in the pipeline captures and processes a dataset element, may be dependent on other tasks in the pipeline, may have a different computational complexity and may be filtered out from progressing in the pipeline. The goal of this work is to develop an efficient scheduler that automatically (i) manages a parallel data reading and an adequate data structure creation, (ii) adaptively defines the most efficient order of pipeline execution of the tasks, considering their inter-dependence and both the filtering out rate and the computational weight, and (iii) manages the parallel execution of the computational tasks in a multicore system, applied to the same or to different dataset elements. A real case study data analysis application from High Energy Physics (HEP) was used to validate the efficiency of this scheduler. Preliminary results show an impressive performance improvement of the pipeline tuning when compared to the original sequential HEP code (up to a 35x speedup in a dual 12-core system), and also show significant performance speedups over conventional parallelization approaches of this case study application (up to 10x faster in the same system).**

Keywords*—*High Throughput Computing, Pipeline, Task Scheduling, Execution Efficiency, Scientific Analyses.

I. Introduction

Scientific software to analyse experimental data to extract useful information (to answer questions, to test hypotheses or to prove theories) are often developed by scientists to automate the analysis of very large datasets. These analyses repeatedly compute a set of tasks, in a given order, with independent datasets and store the relevant results. This set of tasks is structured in a processing pipeline, where each stage may contain a simple or complex computing tasks, followed by an evaluation of a given data property, aiming to discard irrelevant information from going through the rest of the pipeline. This type of scientific code is here addressed as a pipelined scientific data analysis.

The intensive use of these analyses and the increasing complexity of their algorithms require an efficient data processing throughput. Current computer systems are becoming highly parallel even at the processing devices, through increasing values in multiple CPUs per device, from multicore devices (currently up to 18 cores at Intel) to many-core devices (36 tiles of dual cores also at Intel).

Scientists were forced to move into parallel software and their approach to this new paradigm applied to their data analyses, which are considered as embarrassingly parallel applications, typically resorts to conventional parallelization schemes: simultaneous processing different dataset elements. These techniques may improve their data analysis performance and throughput, but not every code scales well in a multicore environment (e.g, when conflicts occur in concurrent memory accesses), and their behaviour is also highly dependent on the computational complexity of the pipeline stages: the code under execution may be compute-bound (more computing resources are required), or memory-bound (better data locality is required and/or wider memory bandwidth) or even I/O-bound (too much data required from secondary storage for so little processing).

Sophisticated mechanisms are required to better explore parallelism. Computer scientists have the responsibility to develop and provide adequate tools for scientists to aid them in the development of efficient parallel code, hiding some of the complexities to deal with parallel computing environments. One such tool is currently under development with this goal: a Highly Efficient Pipelined Framework, HEP-Frame [1], which lets the user provide the raw data and the sequential computational tasks to perform at each pipeline stage and their inter-dependencies, and HEP-Frame automatically builds an efficient data structuring and manages the parallel processing of the data analyses.

The latest version of HEP-Frame did not explore yet the power of these sophisticated mechanisms to better explore parallelism. This paper addresses the development and validation of a scheduler to efficiently manage the parallel data setup and the re-ordering and execution of the pipeline stages in a multicore environment. This scheduler acts at a multi-level stage: to load the raw data into adequate data structures (the data setup), and to improve the execution efficiency through pipeline re-ordering (respecting user-defined dependencies between stages) and load distribution across all available computing resources. To validate the efficiency of HEP-Frame a real case study from High Energy Physic (HEP) was selected, were data collected from sensors at the ATLAS Experiment (CERN) [2] were analysed to prove a theory (the Higgs boson) under three distinct configurations: (i) a memory-bound code, when

the measured data from sensors is assumed 100% accurate, (ii) a compute-bound code, when the measured values are within 99% confidence interval (and over 1k variations for each measured value was considered) and (iii) when the HEP scientist decided to swap two pipeline stages for other two with the same dependencies but different properties (one filters out more elements, the other is computationally heavy).

This communication is structured as follows: section II describes the general structure of pipelined scientific data analyses; section III presents the scheduler and details the different techniques used to explore parallelism and improve application performance; section IV presents three real pipelined scientific data analyses to evaluate the scheduler; section V assesses and evaluates the performance of the scheduler; section refconclusions concludes the communication with a critical analysis and suggestions for future development.

II. Pipelined Scientific Data Analyses

In this communication a scientific data analysis is a process that converts raw scientific data (often from experimental measurements) into useful information to answer questions, test hypotheses or prove theories. When dealing with large amounts of experimental data, data is read from one or more files in variable sized chunks or datasets, and placed into an adequate data structure. When the processing is computational demanding, each dataset element is then processed by a pipeline of propositions: each contains a computational task that usually modifies element properties, may depend from previous propositions and may be followed by an evaluation of a criterion to decide if the element is discarded or processed by the next proposition. Figure 1 shows a typical structure of such scientific data analysis.

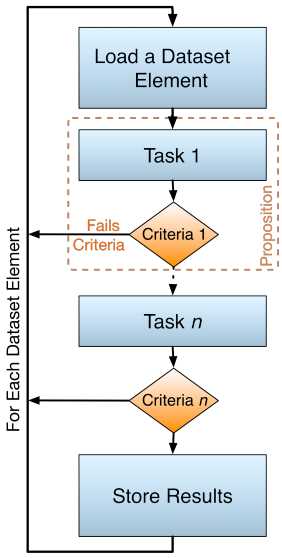

Fig. 1: Structure of a typical flexible pipelined scientific data analysis.

In computational terms, the pipeline processing duration of each dataset element is variable as it may be discarded by a proposition. The execution time of each individual proposition is also dependent on the computational task, whose complexity may vary according to different dataset properties. The default organization of the pipelined propositions, as defined by the scientist, is not guaranteed to be the most efficient, as propositions with long execution times might be placed earlier in the pipeline, while propositions that discard more dataset elements might be executed in the later stages. A formalization of this pipelined is described in depth in [1].

Parallel implementations of these analyses where different threads process different dataset elements are often used in pipelined scientific data analysis, as there are usually no data dependencies among dataset elements. However, this naive approach does not exploit the characteristics of the pipeline, which, if taken into account, may provide significant performance improvements.

III. Scheduling with Pipeline Reordering

Propositions are characterized by their execution time and the amount of data they discard, as explained before. Although the propositions order in the pipeline may have some meaning in the context of the scientific domain, if their interdependencies are respected, reordering the pipeline may lead to a faster analysis execution, by combining two properties of each proposition: their execution time and the amount of data elements they discard. If the propositions that discard more elements are placed earlier in the pipeline and the heavier propositions in later stages, the performance improvements can be substantial.

HEP-Frame has a sequential mechanism to reorder propositions in pipelined scientific data analyses, as presented in [1]. It monitors the weight of propositions at runtime and attempts to reorder the pipeline at given checkpoints. This mechanism solves an NP-Complete problem, the Hamiltonian path [3], which computes the pipeline order with the lowest cost, given the proposition weights and respecting all user-defined dependencies. However, this algorithm had a significant impact in the overall execution time and required a considerable number of iterations to obtain a good pipeline order. A preliminary multithreaded implementation of this algorithm was tested, but the required thread synchronization limited its scalability.

A two-stage pipeline-aware scheduler for multicore environments is proposed in this paper, which aims to efficiently use the available computational resources in multicore CPU devices. The first stage implements a parallel file reading and data structure creation, here considered as the data setup, while the second stage manages the parallel execution of propositions of the same dataset element, the parallel execution of multiple dataset elements, and a soft reordering of the propositions pipeline, as presented in figure 2. This scheduler was implemented in HEP-Frame.

Most scientific data analyses have no data dependencies and are designed to process large amounts of files. Scientific applications usually sequentially read chunks of files and then process the data in parallel. The proposed scheduler reads these files and builds the data structure in parallel, assigning

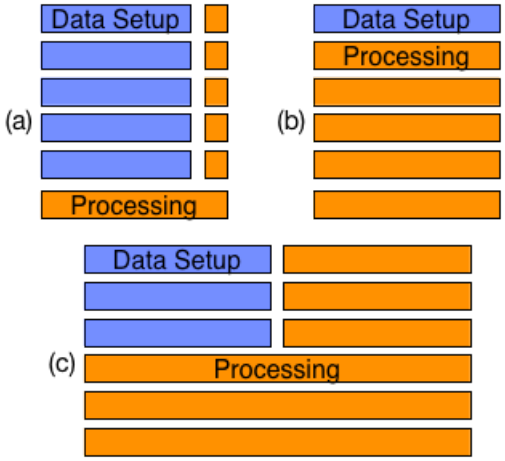

Fig. 2: Scheduler data setup and processing balancing: (a) initial and final configuration when the data setup execution time (*DST*) is much longer than data processing execution time (*DPT*); (b) final configuration when *DPT* is much longer than *DST*; (c) a possible final configuration otherwise.

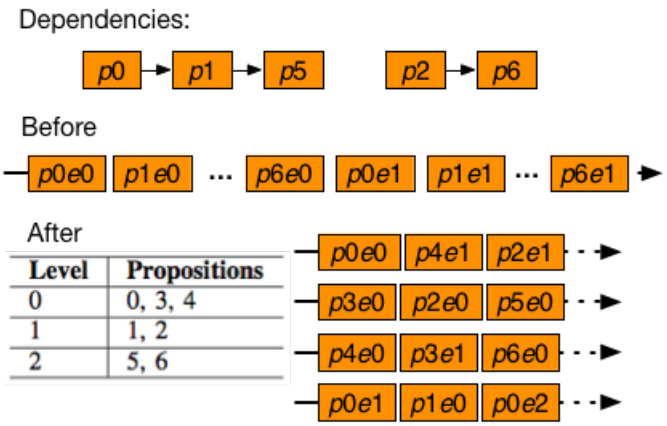

Fig. 3: Sample table for 7 propositions p with the specified dependencies, with their execution before and after using the scheduler for various dataset elements e.

a subset of the files to each thread. The scheduler starts with an initial configuration with a higher amount of threads to perform the data setup (*reader* threads) than threads to process the data (*process* threads). If the processing is much longer than the data setup, the scheduler should allocate more threads to the processing. If the data setup time is similar to the processing, an intermediate configuration should be used. A heuristic to perform this balance at runtime is currently being developed.

The *reader* threads may not be able to load data with the same rate as it is computed by the *process* threads, which leads to starvation. Two alternatives approaches to solve this problem were tested. In the first approach, if a *process* thread does not have data available to process it is put to sleep. *Reader* threads periodically send wake up signals to sleeping *process* threads after loading defined amounts of data. In the second approach, if a *process* thread does not have data available to process it is put to sleep by a defined amount of time. Preliminary tests showed that the second approach provided the best performance as it had a minimal overhead, as it is only required a very small amount of synchronizations among threads, and reduced thread downtime. The amount of time that a *process* thread is sleeping in the first approach is dependent on the time that it takes to load a chunk of data; if the processing is complex it should start as soon as a dataset element is loaded and not after a chunk. This is not an issue if the thread sleeping time is independent on the complexity of the data.

The scheduler stores the propositions identification in a table, as presented in figure 3, which is then used to feed the *process* threads. A directed graph is created when initializing the scheduler, in which propositions are represented as nodes and the dependencies between them as the edges. A standard Breadth-First Search (BFS) algorithm [4] is used to obtain a list of all paths in the graph. In this context, a path represents a chain of dependencies, such as $prop_5$ depends on $prop_1$ that depends on $prop_0$. The longest chain of dependencies is used to create the table, with one table line per proposition. The remaining chains of dependencies are inserted into the table, with the first proposition in the chain in the first line of the table. The propositions without dependencies are also inserted in the first line of the table.

When edges connect all nodes in a graph, the complexity of the BFS algorithm is $O(|N|^2)$. However, the amount of propositions and dependencies in pipelined scientific data analyses is usually not large enough to make the BFS computation a bottleneck (18 propositions and 33 dependency edges for the application presented in section IV). Even though, the BFS algorithm execution time was further reduced by: (i) the scheduler, which stores the dependencies as the user defines them, keeping track of which propositions start a chain of dependencies and (ii) the improved BFS algorithm, which starts to search for chains of dependencies only at the propositions the scheduler defined as starting one chain (the original algorithm searches at every proposition for a chain-starting).

The scheduler assigns the propositions to the *process* threads, one at a time, in the order that they are placed in the table. Moving from one line of the table to the other implies that all previous propositions executed and did not discard the dataset element being processed. This ensures that the dependencies are kept. In the table of figure 3, proposition 6 can only be executed after propositions 2, 3 and 5. This mechanism adds barriers between propositions that do not have dependencies, but it is useful to implement *soft reordering* of the pipeline. If a proposition discards the current dataset element all other threads stop processing and the scheduler assigns propositions relative to the next dataset element.

The number of *process* threads may be higher than the

number of propositions in a given line of the table. Since it is not possible to assign propositions from the next line, the scheduler assigns propositions not yet applied to the next element in the dataset. However, the scheduler may later assign an available proposition of the previous dataset element to the *process* thread. This behaviour is schematized in figure 3. The scheduler has a buffer to keep track of the propositions applied to the each dataset element that started being processed but did not finish execution.

As mentioned before, the pipeline reordering mechanism presented in [1] is not scalable. The proposed scheduler implements a less strict pipeline reordering mechanism based on the presented proposition table layout. The scheduler monitors the propositions execution, assigning each a weight 70% based on the data filtering ratio and 30% on the execution time (these values were obtained through preliminary tests but a dynamic approach is currently being evaluated). The order of the propositions is ensured by the lines of the table, which act as barriers. Propositions should be moved through the different levels of the table, in such a way that lighter propositions, which filter more dataset elements and have a shorter execution time, are placed earlier, while heavier propositions placed in the later lines of the table.

Moving independent propositions inside the table is straight forward as they are only restricted by the availability of lines on top or bottom of them, depending on which way they are moved. However, if the scheduler moves a proposition that is part of a chain of dependencies, all precedent or subsequent propositions of the chain will also be moved, depending on which way the original proposition is moved. An example of this behaviour is presented in figure 4. Lists of precedent and subsequent propositions for each proposition are used to optimize the process of moving a proposition in a chain of dependencies. This avoids traversing the table to check if each proposition is dependent on the one to move. Those lists also store the position of the propositions in the table to reduce access times.

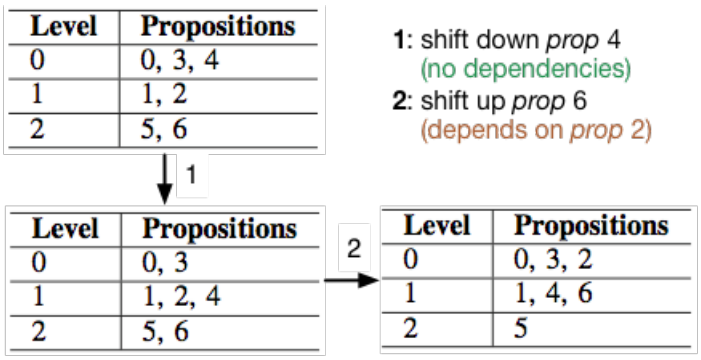

Fig. 4: Proposition table update as the scheduler shifts propositions 4 and 6.

A simple heuristic is used to sort the propositions in the table. Propositions with a weight in the range of 0 to 0.33 should be placed in the first line, weight between 0.33 and 0.66 should be placed in the second line and the heavier propositions should be placed in the third line. Since there are as many table levels as the number of propositions is in the longest dependency chain, the scheduler attempts to move a proposition by only one level if it is not in the correct position. This *soft reordering* of the pipelined is performed at predefined checkpoints (after processing 200 dataset elements).

Thread Building Blocks [5] Flow Graphs have a similar representation of pipeline execution of tasks. This mechanism takes into account dependencies among group of propositions but only extracts parallelism by simultaneous execution of non-dependent propositions. It neither attempts to reorder the pipeline nor extracts parallelism of different dataset elements and propositions execution simultaneously. While it may perform well on a wider set of applications it was not fully adapted to the requirements of pipelined scientific data analysis.

A similar optimization of the pipeline approach to task execution was also presented in [6], but targeting a wider range of applications. In this work, the authors did not take into account that the pipeline in scientific data analyses may discard dataset elements and only focused on the parallelization of tasks, rather than the pipeline reordering. An out-of-order execution of the tasks is performed to diminish the time that the CPU cores idle due to data dependencies among tasks, in a similar way to instruction level parallelism.

IV. The Case Studies

HEP scientists at the ATLAS Experiment [2] at CERN developed a scientific data analysis code, the $t\bar{t}H$ analysis, to study the associated production of top quarks with the Higgs boson [5], following head-on proton-proton collisions (known as events) at the Large Hadron Collider (LHC). This code was selected to validate the HEP-Frame scheduler and to evaluate its performance impact. Figure 5 represents the final state topology of a proton beam collision for the $t\bar{t}H$ production.

The final state of an event is recorded by the ATLAS particle detector, which measures the characteristics of the bottom quarks (detected as jets of particles due to a hadronization process) and leptons (both muons and electrons), but not the neutrinos, as they do not interact with the detector sensors.

$t\bar{t}H$ analytically computes the characteristics of the neutrinos with known information, to reconstruct both top quarks and the Higgs boson. This process, known as kinematical reconstruction, tests every combination of bottom quarks and leptons (there may be more than needed due to other background interactions among particles), which are stored in a specific structure in pre-defined files provided by the experiments at the LHC. $t\bar{t}H$ may sample the kinematical reconstruction process within the 99% confidence level, to reduce the relative measurement uncertainty of the sensors in the ATLAS Experiment detector, which has a direct impact on the complexity of the performed computation per event.

$t\bar{t}H$ has 18 stages to perform a computational task and to filter out measured events that do not comply with the

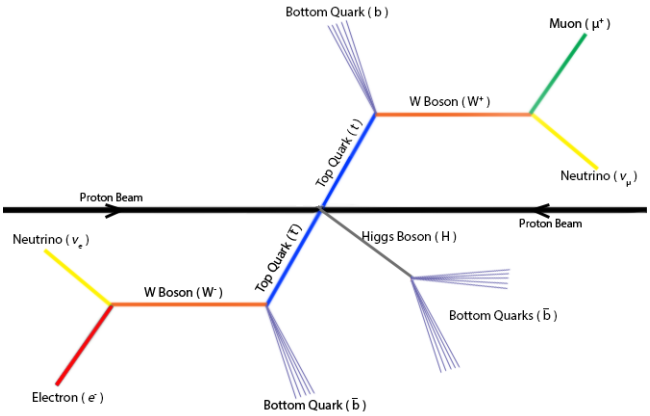

Fig. 5: Schematic representation of the $t\bar{t}$ system and Higgs boson decay.

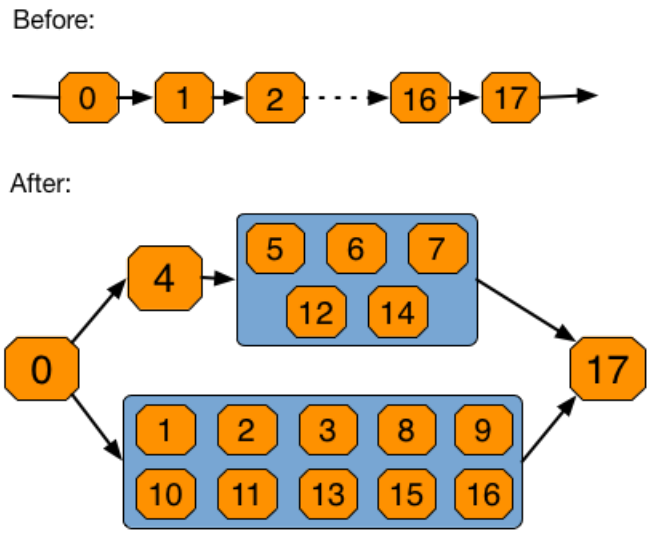

Fig. 6: Schematic representation of the possible parallel proposition execution before and after using the scheduler, respecting the dependencies in the $t\bar{t}H$ applications.

theoretical model expectations, each coded as a proposition in HEP-Frame, with the dependencies specified in figure 5. In the original analysis code their id number represented their execution order in figure 6 (see "Before"). The propositions inside the blue boxes do not have dependencies among them but depend, as a group, on other propositions.

Three versions of the $t\bar{t}H$ analysis were considered as representative case studies: (i) one considers that the data measured by the ATLAS detector is 100% accurate when reconstructing the event, `ttH_as` (accurate sensors), a memory-bound code in most computing systems; (ii) another considers that the ATLAS detector has a measurement accuracy error up to $\pm 1\%$ and performs an extensive sampling within the 99% confidence interval when reconstructing the event, named `ttH_sci` (sensors with a confidence interval), from which only the best reconstruction is considered; this version performs 1024 samples: each requires the generation of 30 different pseudo-random numbers, to a total of approximately 30K numbers per event, leading to a compute-bound code; (iii) the third version `ttH_scinp` (`sci` with a new pipeline), replaces propositions 13 and 16 to perform different operations on the data element, maintaining the same overall proposition dependencies and the same sampling of the confidence interval of `ttH_sci`; this version is also compute-bound.

The default organization of the pipelines in these analyses was setup by the HEP scientist that developed the code and it was already optimized under his point of view: the heavier proposition is the last pipeline stage, while previous stages filter out a significant percentage of events (dataset elements).

The 18 `ttH_as` propositions have execution times always shorter than 104 nanoseconds, of which 16 pass more than 90% of the events. Two propositions have a passing ratio of 63% and 50%, respectively. `ttH_sci` has the same filtering ratios, since it shares the same pipeline as `ttH_as`, but has two heavier propositions with an execution time of 106 and 108 nanoseconds, respectively. The new proposition 13 in `ttH_scinp` has a longer execution time than in `ttH_sci`, around 108 nanoseconds, and proposition 16 has now a passing ratio of 30%, versus 99% in `ttH_sci`. Tables I and II summarize this information.

TABLE I: Characterization of the 18 propositions in the $t\bar{t}H$ applications.

	Execution Time (nanoseconds)			
	$[10^8, 10^6[$	$[10^6, 10^4[$	$[10^4, 10^3[$	$[10^3, 0[$
`ttH_as`	0	0	1	17
`ttH_sci`	1	0	1	16
`ttH_scinp`	2	0	1	15

The parallel implementations of these three analyses followed two distinct conventional parallelization approaches for embarrassingly parallel problems, often used in scientific computing. The first approach uses a sequential code for the data setup and a parallel approach to process events from separate files at each core of a shared memory environment at a multicore system; each thread runs a sequential version of the full pipeline, repeating this process for each input file (here addressed as *S+MT*, Sequential+MultiThreading). This approach simply uses OpenMP [7]. The second approach uses two sets of parallel processes, one for the data setup and the other to process the events, in a distributed memory environment at the same multicore system (addressed as *MP*, MultiProcessing). Both approaches are illustrated in figure 7.

TABLE II: Characterization of the 18 propositions in the $t\bar{t}H$ applications.

	Passing ratios (% of passing dataset elements)			
	[100%, 99%[	[99%, 90%[	[65%, 60%[	[50%, 0%[
`ttH_as`	15	1	1	1
`ttH_sci`	15	1	1	1
`ttH_scinp`	14	1	1	2

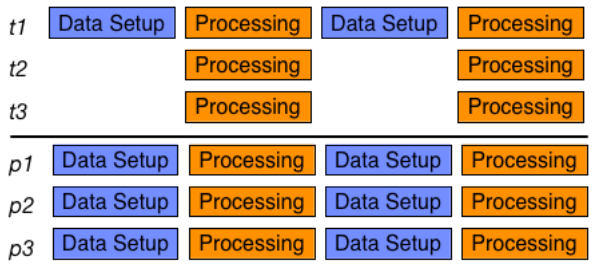

Fig. 7: Schematic representation of the conventional *S+MT* (top) and *MP* (bottom) parallelizations.

V. Performance Results and Discussion

The testbed used for the quantitative evaluation of the HEP-Frame with the two-stage scheduler was a dual socket cluster-computing node with twelve-core Intel Xeon E5-2695v2 Ivy Bridge CPU devices at 2.4GHz [8]. The three configurations of the case study, as described before, were `ttH_as`, `ttH_sci` and `ttH_scinp`. A k-best measurement heuristic [9] was used to ensure that the results can be replicated, with $k = 5$, a minimum of 15 measurements and a maximum of 25. The scheduler was tested using 6, 12 and 24 cores (1 thread/core). The $t\bar{t}H$ analyses were tested with 128 files, each with $\pm$6000 events. Preliminary tests suggested that splitting in half the total number of threads to data setup and processing provided an acceptable default configuration to test these scientific analyses.

Figure 8 shows the impact of the scheduler on the speedup of `ttH_sci` and `ttH_scinp`, when compared to their sequential execution. Since the order of the pipeline in the `ttH_sci` is close to its optimum configuration the maximum achieved speedup is only 16x for 24 threads, due to the parallel data setup and parallelization of the proposition execution. However, the performance of `ttH_scinp` is improved by a factor of 39 for 24 threads. The superscalar speedups of `ttH_scinp`, for every thread count tested, is due to the combination of the parallel data setup with the pipeline reordering, which is less optimized than on `ttH_sci` by default. In this particular case, proposition 13 is placed at the end of the pipeline and proposition 16 is moved to its initial stages.

Figures 9 and 10 show the comparison between the scheduler and two conventional parallelization schemes for scientific applications presented in section IV, *S+MT* and *MP* respectively. The scheduler outperforms the *S+MT* parallelization up to 7x for the `ttH_as` application, as it benefits from the parallel data setup due to the small amount of computation performed per dataset element, and up to 10x for the `ttH_scinp` application, as the pipeline reordering has a significant impact on this application performance. The scheduler also outperforms the more complex *MP* parallelization, but with smaller performance improvements: both `ttH_as` and `ttH_scinp` had speedups of 3.5 and 4.2, respectively.

Hardware multithreading (Intel Hyperthreading) did not pro-

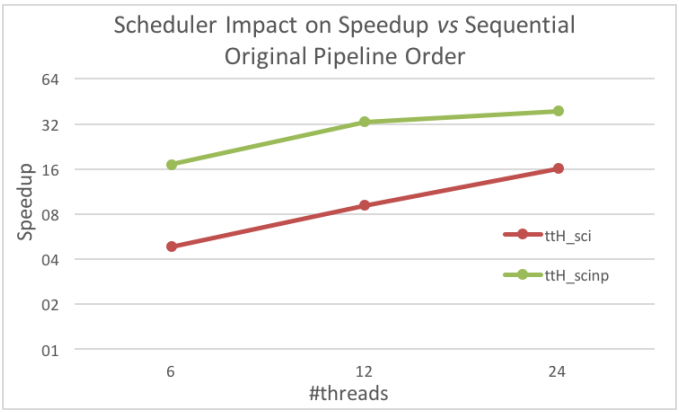

Fig. 8: Speedup comparison between the proposed scheduler and the sequential implementation of the $t\bar{t}H$ application.

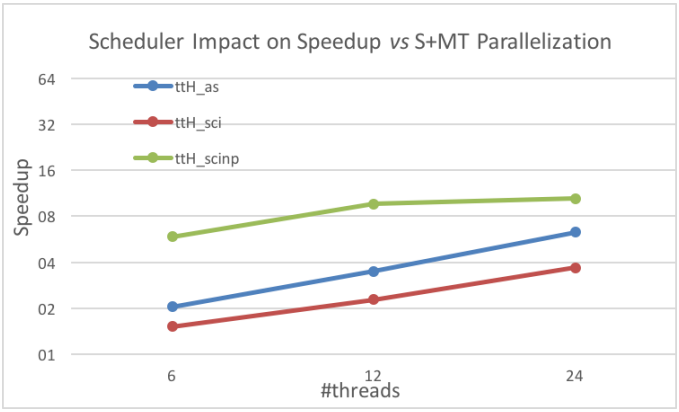

Fig. 9: Speedup comparison between the proposed scheduler and a conventional *sequential + multithreading* parallelization in HEP-Frame.

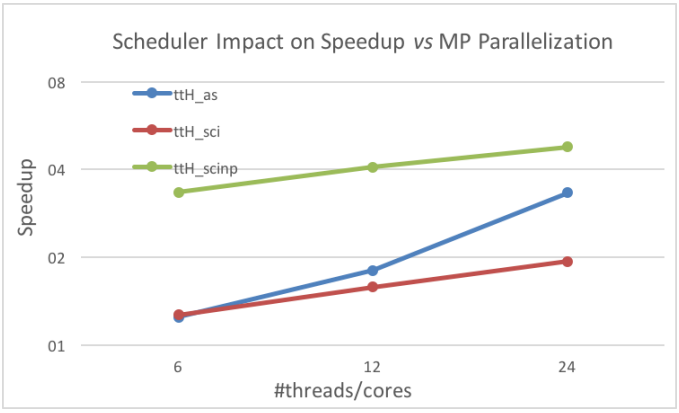

Fig. 10: Speedup comparison between the proposed scheduler and a conventional *multiprocess* parallelization in HEP-Frame.

vide significant performance gains for any of the tested analyses. The processing throughput of the sequential `ttH_as`, `ttH_sci` and `ttH_scinp` was 557, 79 and 572 dataset elements per second, and was improved to 17478, 1617 and 22245 dataset elements per second. With the scheduler, the cost of sampling 1024 values within the confidence interval for each of the 30 sensor data for each event in `ttH_sci` is only a decrease in throughput by a factor of less than 11, which greatly improves the quality of the results over `ttH_as`.

As mentioned before, the higher speedups achieved for the `ttH_as` application indicate that the scheduler works best for memory-bound applications due to the parallel data setup. In this case, the peak memory bandwidth increased from 3.5 GB/s, in the *S+MT* parallelization, to 4.9GB/s using the scheduler for 24 threads. An analysis of the scheduler with the Intel VTune profiler [10] showed that *process* threads were waiting for data to be loaded less than 10% of the overall data setup time. It also showed that the overhead of the pipeline reordering was less than 5% for `ttH_as` and less than 1% for both `ttH_sci` and `ttH_scinp`, for 24 threads.

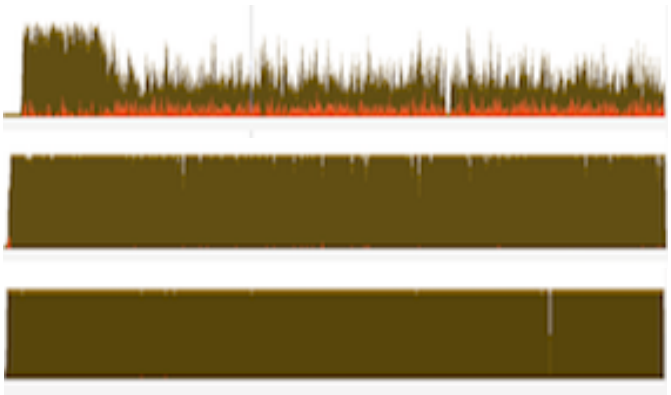

Fig. 11: CPU usage with 24 threads and overall execution time: `ttH_as` (top, $54s$), `ttH_sci` (middle, $580s$), and `ttH_scinp` (bottom, $120s$).

Figure 11 shows the CPU usage of the three applications obtained with Intel VTune. While `ttH_sci` and `ttH_scinp` use 100% of the CPU almost all the time, due to their compute-bound nature, `ttH_as` only has high CPU usage when combining the data setup with proposition processing, as it heavily relies on how fast the CPU cores get the data from memory. This analysis suggests that there is still room to improve the scheduler efficiency: the heuristic to balance *reader* threads with *processing* threads needs to be refined.

VI. Conclusions and Future Work

This communication describes a two-stage scheduler for pipelined scientific data analyses, where large raw experimental data is converted into useful information through complex computational tasks: (i) it implements a parallel reading of input files and building of an adequate data structure; and (ii) it manages the parallel execution of propositions of the same dataset element, the parallel execution of multiple dataset elements, and a soft reordering of the propositions pipeline to filter out most of the elements at the earliest time.

The scheduler was implemented into HEP-Frame, a framework under development to aid the efficient execution of scientific code. This code is developed by the end user, who may not have explored the potential of code vectorization; current version of HEP-Frame may only aid in compiling the code with this flag on, together with a report generation of vectorization results, and let the scientist improve the code performance if she/he wishes.

The scheduler was validated with a real case study from HEP scientists: the $t\bar{t}H$ particle physics event data analysis, developed and used by CERN researchers in a production environment, with 18 propositions in the pipeline. Three versions of $t\bar{t}H$ were selected: a configuration used for fast preliminary event analysis, `ttH_as` (assuming accurate sensors), with a memory-bound behaviour; a configuration oriented to perform a more extensive analysis of the events, `ttH_sci` (1024 variations within the sensor confidence interval), which is compute-bound; a configuration that performs an extensive analysis using different filtering criteria and computation in two pipelined propositions, `ttH_scinp` (`sci` with new pipeline propositions), which also behaves as compute-bound.

The scheduler performance on up-to-24 Xeon cores in cluster node was compared with a sequential configuration of the two compute-bound $t\bar{t}H$ versions with a static pipeline order, defined by the user common sense. It showed superscalar speedups (up to 39x), for every measured thread configuration with `ttH_scinp`, as the parallel data setup and reordering of the pipeline had a significant impact on performance. The scheduler achieved a speedup of 16x for the `ttH_sci`, which had an almost optimum pipeline order by default.

The scheduler performance on the same cluster node was also compared with two conventional parallel approaches to the same data analysis: (i) a sequential data setup with multithreaded dataset processing (*S+MT*) and (ii) a multiprocess execution with two sets of parallel activities: the data setup and the event processing (*MP*). It outperformed both parallel implementations, with speedups up to 10x for the `ttH_scinp` application, where the pipeline benefited the most from reordering.

These performance outcomes were obtained against a set of analyses that were already coded with a reasonable default pipeline order. The scheduler in HEP-Frame has the potential for higher performance improvements if the defined analysis pipeline order is computationally unreasonable: it only requires from the user the set of input files, the sequential code for each proposition and the list of inter-dependencies among the propositions. It then automatically manages the parallel data setup and the propositions execution.

The feasibility of offloading suitable pipeline propositions to manycore coprocessors is currently being evaluated. The goal is to improve the scheduler to simultaneously process propositions in the multicore CPU devices and in manycore coprocessor devices. It is not expected that current manycore coprocessors will outperform these computations faster than

multicore devices, but next generation of such devices are still an opportunity that deserves our attention.

ACKNOWLEDGMENT

This work was supported by FCT (Fundação para a Ciência e Tecnologia) within Project Scope (UID/CEC/00319/2013), by LIP (Laboratório de Instrumentação e Física Experimental de Partículas) and by Project Search-ON2 (NORTE-07-0162-FEDER-000086), co-funded by the North Portugal Regional Operational Programme (ON.2 - O Novo Norte), under the National Strategic Reference Framework, through the European Regional Development Fund.

REFERENCES

[1] A. Pereira, A. Onofre, and A. Proença, "HEP-Frame: A Software Engineered Framework to Aid the Development and Efficient Execution of Scientific Code," *International Computational Science and Computational Intelligence – CSCI 2015*, 2015.

[2] T. A. Collaboration, "The atlas experiment at the cern large hadron collider," *Journal of Instrumentation*, 2008.

[3] E. W. Weisstein, "Hamiltonian Path," http://mathworld.wolfram.com/HamiltonianPath.html.

[4] S. S. Skiena, *The Algorithm Design Manual*, 2nd ed. Springer-Verlag London, 2008.

[5] Intel, "Threading Building Blocks (Intel TBB) flow graphs." [Online]. Available: https://www.threadingbuildingblocks.org/tutorial-intel-tbb-flow-graph

[6] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero, "Task superscalar: An out-of-order task pipeline," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 89–100. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2010.13

[7] O. A. R. Board, "Openmp Application Program Interface," OpenMP Architecture Review Board, Tech. Rep., 2013.

[8] Intel, "Intel xeon processor e5 v2 family: Datasheet," Intel Corporation, Tech. Rep., 2013.

[9] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 1st ed. Prentice Hall, 2003.

[10] Intel, "Intel VTune Amplifier XE 2016 and Intel VTune Amplifier 2016 for Systems Help," Intel, Tech. Rep., 2016.